Divergence

We learned that if S in σ converges to τ, we have $\langle S, \sigma\rangle \to^* \langle E, \tau\rangle$ and $M(S,\sigma) = \{\tau\}$; but what if S diverges?

- We define pseudo-state "$\perp$" (reads "bottom") to represent a program cannot terminate successfully.
  - $\perp$ is not a real state in memory.
  - There can be multiple reasons a program cannot terminate successfully, such as the program diverges, or the program meets some runtime error and halts.

- Denotationally, we write $M(S,\sigma) = \{\perp_d\}$ to represent S diverges in σ. Operationally, $\langle S, \sigma\rangle \to^* \langle E, \perp_d\rangle$ means that S starting in σ diverges.

- Here, we present two situations that we can say S diverges in σ. However, if program S and state σ doesn't satisfy the following situation, it is still possible that S diverges in σ; in general, "Whether arbitrary S converges in arbitrary σ?" is an undecidable problem.

  - In the sequence of configurations $\langle S_0, \sigma_0\rangle \to \langle S_1, \sigma_1\rangle \to \langle S_2, \sigma_2\rangle \to \cdots$, if $\exists i. \exists j. i \neq j \wedge S_i = S_j \wedge \sigma_i = \sigma_j$, then S_0 starting in σ_0 diverges.

1. Evaluate $W \equiv \textbf{while } T \textbf{ do skip od}$ in σ.
   Since $\langle W, \sigma\rangle \to \langle \textbf{skip}; W, \sigma\rangle \to \langle W, \sigma\rangle$, thus $M(W, \sigma) = \{\perp_d\}$. Or operationally, we can write $\langle W, \sigma\rangle \to^* \langle E, \perp_d\rangle$.

   - While evaluating an iterative statement $W \equiv \textbf{while } B \textbf{ do } S \textbf{ od}$, and get a sequence $\langle W, \sigma_0\rangle \to^* \langle W, \sigma_1\rangle \to^* \langle W, \sigma_2\rangle \to \cdots$, if we can prove that $\neg\exists i. \sigma_i(B) = F$, then W in σ_0 diverges.

2. Calculate $M(W, \sigma_0)$ where $W \equiv \textbf{while } x \neq n \textbf{ do } x := x - 1 \textbf{ od}$ and $\sigma_0 = \{x = -1, n = 0\}$.

$$\begin{aligned} M(W, \sigma_0) &= M(W, \sigma_0 = \{x = -1, n = 0\}) \\ &= M(W, \sigma_1 = \sigma_0[x \mapsto -2] = \{x = -2, n = 0\}) \\ &= M(W, \sigma_2 = \sigma_1[x \mapsto -3] = \{x = -3, n = 0\}) \\ &= \cdots \end{aligned}$$

   - We start with $\sigma_0(x) < \sigma_0(n)$; after each iteration of W, the value bind to x can only be updated to a smaller number and the value of n never changes. Thus $\forall i \geq 0. \sigma_i(x) < \sigma_i(n)$, and $M(W, \sigma_0) = \{\perp_d\}$.

Runtime Errors

- Like divergence, we use a pseudo-state (also a pseudo-value) "$\perp_e$" to represent the state (or value) when a runtime error happens.
- Runtime errors can happen in both expressions and programs. Let's look at runtime errors in expressions first. We write $\sigma(e) = \perp_e$ to represent that the evaluation of expression e in state σ causes a runtime error. Here "$\perp_e$" is a pseudo-value, which means it is not a real value, it is used to represent values such as $x/0$ or $sqrt(-1)$.
  - If e might cause runtime error, then instead of $\sigma(e) \in V$ for some set value set V, we now have $\sigma(e) \in V \cup \{\perp_e\}$ as the range of $\sigma(e)$. For example, since x/y might cause a runtime error, then its value in some state σ might result in $\perp_e$, thus we write $\sigma(x/y) \in \mathbb{Z} \cup \{\perp_e\}$.

- Types of runtime errors:
  - **Primary Failures:** The primitive values and operations being supported determine some set of basic runtime errors.
    - Array index out of bounds: $\sigma(b[e]) = \perp_e$ if $\sigma(e) < 0$ or $\sigma(e) \geq size(b)$. Similar situation for multi-dimensional arrays.
    - Division by zero: $\sigma(e_1/e_2) = \sigma(e_1 \,\%\, e_2) = \perp_e$ if $\sigma(e_2) = 0$.
    - Square root of negative number: $\sigma\left(sqrt(e)\right) = \perp_e$ if $\sigma(e) < 0$.

  - **Hereditary Failure**: If evaluating a subexpression fails, then the overall expression fails.
    - If op is a unary operator, then $\sigma(op\ e) = \perp_e$ if $\sigma(e) = \perp_e$. For example: $e \equiv x/0$, then $\sigma(e) =\perp_e$, and $\sigma\big(sqrt(e)\big) =\perp_e$ as well.
    - If op is a binary operator, then $\sigma(e_1\ op\ e_2) = \perp_e$ if $\sigma(e_1) =\perp_e$ or $\sigma(e_2) =\perp_e$.
    - For a conditional expression, $\sigma(\mathbf{if}\ B\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\ \mathbf{fi}) =\perp_e$ if one of the following three situations occurs:
      1) $\sigma(B) =\perp_e$
      2) $\sigma(B) = T$ and $\sigma(e_1) = \perp_e$
      3) $\sigma(B) = F$ and $\sigma(e_2) = \perp_e$

3. What are the states that will cause runtime errors for each of the following expressions?
   a. $b[x/y]$ Some state σ where $\sigma(y) = 0$, or $\sigma(x/y) < 0$,or $\sigma(x/y) \geq size(b)$.
   b. $sqrt(x) + sqrt(x/y)$ Some state σ where $\sigma(x) < 0$, or $\sigma(y) = 0$, or $\sigma(x/y) < 0$.
   c. $\mathbf{if}\ y = 0\ \mathbf{then}\ 0\ \mathbf{else}\ x/y\ \mathbf{fi}$ No such states.

- A runtime error in an expression can cause the statement it appears in to halt unsuccessfully. We write $\langle S, \sigma\rangle \to \langle E, \perp_e\rangle$ for the operational semantics of such a statement.
  - Here $\perp_e$ is a pseudo-state not a pseudo-value. Decide whether $\perp_e$ is a pseudo-state or pseudo-value from the context.

- With runtime errors, let's expand our operations semantics rules:
  - $\sigma(e) = \perp_e$, then $\langle v \coloneqq e, \sigma\rangle \to \langle E, \perp_e\rangle$.
  - $\sigma(b[e_1]) =\perp_e$ or $\sigma(e_2) =\perp_e$, then $\langle b[e_1] \coloneqq e_2, \sigma\rangle \to \langle E, \perp_e\rangle$.
  - If $\sigma(B) =\perp_e$, then $\langle \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}, \sigma\rangle \to \langle E, \perp_e\rangle$.
    - On the other hand, if $\sigma(B) \neq\perp_e$, we will continue to evaluate configuration $\langle S_1, \sigma\rangle$ or $\langle S_2, \sigma\rangle$.
  - If $\langle S_1, \sigma\rangle \to \langle E, \perp_e\rangle$ then $\langle S_1; S_2, \sigma\rangle \to \langle E, \perp_e\rangle$.
  - If $\sigma(B) =\perp_e$, then $\langle \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}, \sigma\rangle \to \langle E, \perp_e\rangle$.
    - On the other hand, if $\sigma(B) \neq\perp_e$, we will continue to evaluate configuration $\langle E, \sigma\rangle$ or $\langle S\colon \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}, \sigma\rangle$.

4. Evaluate $S \equiv x \coloneqq 0; y \coloneqq 1; W$ where $W \equiv \mathbf{while}\ x/y \geq 0\ \mathbf{do}\ y \coloneqq sqrt(y) - 1\ \mathbf{od}$ in state σ.

$$
\begin{aligned}
\langle S, \sigma\rangle \quad &\to^* \langle W, \sigma[x \mapsto 0][y \mapsto 1]\rangle \\
&\to^* \langle W, \sigma[x \mapsto 0][y \mapsto 0]\rangle && \text{//since } \sigma[x \mapsto 0][y \mapsto 1](x/y \geq 0) = T \\
&\to \langle E, \perp_e\rangle && \text{//since } \sigma[x \mapsto 0][y \mapsto 0](x/y \geq 0) = \perp_e
\end{aligned}
$$

<u>Properties and Consequences of ⊥</u>

- $\perp$ refers generically to $\perp_d$ and/or $\perp_e$. We use $\langle S, \sigma\rangle \to^* \langle E, \perp\rangle$ when it's not important which of $\perp_d$ or $\perp_e$ can occur. Similarly, $\perp \in M(S, \sigma)$ means $\langle S, \sigma\rangle$ leads to $\perp_d$ or $\perp_e$.

- Since we use $\perp$ somewhere an actual memory state appears in evaluating a program, we want to look at other situations where a state appears in and think about whether $\perp$ can be used there.
    - $\perp$ is not a well-formed state.
    - **When we say, "for all states…", "for some state…", or "for all semantic values…", "for some semantic values…", we don't include $\perp$.**
    - We cannot add a binding to $\perp$: $\perp \cup \{v = \beta\} = \perp$.
    - Consider $\perp$ as a pseudo-value, then binding it with a variable will result in a pseudo-state: $\sigma(v) \neq \perp$ and $\sigma[v \mapsto \perp] = \perp$. In other words, we cannot bind a variable with pseudo-value $\perp$.
    - Evaluating a variable or an expression in $\perp$ results in a pseudo-value $\perp$: If $\sigma = \perp$ then $\sigma(v) = \sigma(e) = \perp$. In other words, we cannot take the value of a variable or expression in $\perp$.
    - Operationally, execution halts as soon we generate $\perp$ as a pseudo-state: $\langle S, \perp \rangle \to^0 \langle E, \perp \rangle$.
    - Denotationally, we can't run a program in pseudo-state $\perp$: $M(S, \perp) = \{\perp\}$.

(Logic with $\perp$)

- $\perp$ cannot satisfy any predicate: $\perp \nvDash p$ for all p, even if p is the constant T. In general, we now have **three possibilities for a state trying to satisfy a predicate**: $\sigma \vDash p$, $\sigma \vDash \neg p$, or $\sigma(p) = \perp$.
    - Previously we have $\sigma \nvDash p$ means $\sigma \vDash \neg p$, but this is no longer true when $\perp$ is taken into consideration. $\sigma \nvDash p$ means "It is not true that $\sigma \vDash p$" and it means $\sigma \vDash \neg p$ or $\sigma(p) = \perp$ (in other words, $\sigma \nvDash p \Leftrightarrow \sigma \vDash \neg p \vee \sigma(p) = \perp$) now.

5. True or False.

a.	If $\sigma \vDash p$, then $\sigma \nvDash \neg p$.	True.
b.	If $\sigma \vDash p$, then $\sigma(p) \neq \perp$.	True.
c.	If $\sigma \vDash \neg p$, then $\sigma \nvDash p$.	True.
d.	If $\sigma \nvDash \neg p$, then $\sigma \vDash p$.	False.
e.	If $\sigma(p) = \perp$, then $\sigma(\neg p) = \perp$.	True.
f.	If $\sigma(p) \neq \perp$, then $\sigma \nvDash p \Leftrightarrow \sigma \vDash \neg p$	True.
g.	If p is a valid predicate (in other words, $\vDash p$), then $\perp \vDash p$.	False.
h.	If $\nvDash p$, then there exists some σ with $\sigma(p) = \perp$.	False. It means $\exists \sigma . \sigma(p) = \perp \vee \sigma(p) = F$

(Satisfaction by a collection of states)

- We usually use $\Sigma_\perp = \Sigma \cup \{\perp\}$, where Σ is the collection of all (well-formed) states. Let $\Sigma_0 \subseteq \Sigma_\perp$, then we say $\Sigma_0 \vDash p$, if every state in $\Sigma_0 \vDash p$.

6. True or False.

a.	Let $\sigma \in \Sigma$, then $\sigma \neq \perp$.	True
b.	Let $\Sigma_0 \subseteq \Sigma_\perp$, then $\perp \in \Sigma_0$.	False, $\perp$ is allowed to be in such Σ_0, but not necessarily.
c.	Let $\Sigma_0 \vDash p$, then $\perp \notin \Sigma_0$	True, $\perp$ doesn't satisfy any predicate.
d.	$\Sigma_0 \nvDash p \Leftrightarrow \exists \tau \in \Sigma_0 . \tau \nvDash p$	True
e.	If $\perp \in \Sigma_0$, then $\Sigma_0 \vDash \neg p$	False. If $\perp \in \Sigma_0$, then $\Sigma_0 \nvDash p$ and $\Sigma_0 \nvDash \neg p$
f.	$\emptyset \vDash p$ (an empty set of states)	True, "every state" in $\emptyset$ satisfies p
g.	$\Sigma_0 \vDash p \wedge \Sigma_0 \vDash \neg p$ if and only if $\Sigma_0 = \emptyset$	True
h.	Let $\Sigma_0 - \perp = \{\tau\}$, then $\Sigma_0 - \perp \vDash p$ or $\Sigma_0 - \perp \vDash \neg p$	FALSE, it is possible that $\tau(p) = \perp$.
i.	If all state in $\Sigma_0 - \perp$ can evaluate p to either T or F, then $\Sigma_0 - \perp \vDash p$ or $\Sigma_0 - \perp \vDash \neg p$	False, it is possible that some states in $\Sigma_0 - \perp$ satisfy p and some satisfy $\neg p$, then $\Sigma_0 - \perp \nvDash p$ and $\Sigma_0 - \perp \nvDash \neg p$. (I added an assumption, so it doesn't lose nuance.)

Nondeterminism

- Nondeterminism is a theoretical idea, in general it means "don't make decision, consider all possible outcomes at the same time with same probability." Note that, it doesn't mean "randomly pick possible outcome". Just like Schrodinger' cat, it is both alive and dead at the same time with a 50% 50% probability.
    - For example, when we say, "choose one side of a coin nondeterministically", it means "choose head with 50% probability and choose tail with 50% probability." It doesn't "pick one side randomly" because it will result in either a head or a tail.
        - However, a nondeterministic program can be simulated by "random picking a branch among all possible branches for a lot of times". For example, you can simulate the procedure "choosing one side of a coin nondeterministically" by tossing a fair coin a lot of times, then you will come up with a set of outputs:$\{head,\ tail\}$, and the probability of two outputs will be near 50% and 50%.

    - Here is another example. If we have a simple program that returns the maximum between x and y:

      $$\textbf{if } x \leq y \textbf{ then } max \coloneqq y \textbf{ else } max \coloneqq x \textbf{ fi}$$

      This $\textbf{if} - \textbf{else}$ statement is deterministic. When we have $x = y$, it will always choice y to be the max. Consider a different program that does the following:

      "If $x \geq y$ then $max \coloneqq x$, if $x \leq y$ then $max \coloneqq y$"

      and it can evaluate both branches at the same time. When we have $x = y$, then max should have 50% chance to be x and 50% chance to be y.

- Actually, a machine/program cannot run *nondeterministically*, but designing a nondeterministic machine/program is useful:
  i. A nondeterministic machine/program has a deterministic machine/program that can do the same job (although the deterministic version might need to finish the same job using much longer time). This is a topic in CS530 Computational Theory.
  ii. The design process can be simplified. We can avoid some insignificant choice-making and only focus on the important decisions in the design. The design of a nondeterministic machine is easier to read and understand.